Polly's Polyhedral Scheduling in the Presence of Reductions

Johannes Doerfert, Kevin Streit, and Sebastian Hack
Computer Science Department
Saarland University
Saarbrücken, Germany
<lastname>@cs.uni-saarland.de

Zino Benaissa
Qualcomm Innovation Center
San Diego
California, USA
zinob@quicinc.com

ABSTRACT

The polyhedral model provides a powerful mathematical abstraction to enable effective optimization of loop nests with respect to a given optimization goal, e.g., exploiting parallelism. Unexploited reduction properties are a frequent reason for polyhedral optimizers to assume parallelism prohibiting dependences. To our knowledge, no polyhedral loop optimizer available in any production compiler provides support for reductions. In this paper, we show that leveraging the parallelism of reductions can lead to a significant performance increase. We give a precise, dependence based, definition of reductions and discuss ways to extend polyhedral optimization to exploit the associativity and commutativity of reduction computations. We have implemented a reduction-enabled scheduling approach in the Polly polyhedral optimizer and evaluate it on the standard Polybench 3.2 benchmark suite. We were able to detect and model all 52 arithmetic reductions and achieve speedups up to 2.21× on a quad core machine by exploiting the multidimensional reduction in the BiCG benchmark.

Categories and Subject Descriptors

D 3.4 [**Programming languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms; Performance

Keywords

Compiler Optimization; Affine Scheduling; Reductions

1. INTRODUCTION

Over the last four decades various approaches [10, 11, 2, 17, 22, 6, 18, 21, 27, 31, 7] were proposed to tackle reductions: a computational idiom which prevents parallelism due to loop carried data dependences. An often used definition for reductions describes them as an associative and commutative computation which reduces the dimensionality of a set of input data [16]. A simple example is the array sum depicted in Figure 1a. The input vector `A` is reduced to the scalar variable `sum` using the associative and commutative operator +. In terms of data dependences, the loop has to be computed sequentially because a read of the variable `sum` in iteration $i+1$ depends on the value written in iteration i. However, the associativity and commutativity of the reduction operator can be exploited to reorder, parallelize or vectorize such reductions.

While reordering the reduction iterations is always a valid transformation, executing reductions in a parallel context requires additional "fix up". Static transformations often use privatization as fix up technique as it works well with both small and large parallel tasks. The idea of privatization is to duplicate the shared memory locations for each instance running in parallel. Thus, each parallel instance works on a private copy of a shared memory location. Using the privatization scheme we can vectorize the array sum example as shown in Figure 1b. For the shared variable `sum`, a temporary array `tmp_sum`, with as many elements as there are vector lanes, is introduced. Now the computation for each vector lane uses one array element to accumulate intermediate results unaffected by the computations of the other lanes. As the reduction computation is now done in the temporary array instead of the original reduction location we finally need to accumulate all intermediate results into the original reduction location. This way, users of the variable `sum` will still see the overall sum of all array elements, even though it was computed in partial sums first.

```
for (i = 0; i < 4 * N; i++)
  sum += A[i];
```

(a) Sequential array sum computation.

```
tmp_sum[4] = {0,0,0,0}
for (i = 0; i < 4 * N; i+=4)
  tmp_sum[0:3] += A[i:i+3];

sum += tmp_sum[0] + tmp_sum[1];
     + tmp_sum[2] + tmp_sum[3];
```

(b) Vectorized array sum computation.

Figure 1: A canonical example of a single address reduction.

IMPACT 2015
Fifth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

http://impact.gforge.inria.fr/impact2015

Transformations as described above have been the main interest of reduction handling approaches outside the polyhedral world. Associativity and commutativity properties are used to extract and parallelize the reduction loop [10, 6] or to parallelize the reduction computation with regards to an existing surrounding loop [17, 18, 21, 27, 31]. While prior work on reductions in the polyhedral model [22, 23, 24, 9, 32] was focused on system of affine recurrences (SAREs), we look at the problems a production compiler has to solve when we allow polyhedral optimizations that exploit the reduction properties. To this end our work supplements the polyhedral optimizer *Polly* [8], part of the *LLVM* [14] compiler framework with awareness for the associativity and commutativity of reduction computations. While we are still in the process of upstreaming, most parts are already accessible in the public code repository.

The contributions of this paper include:

- A powerful algorithm to identify reduction dependences, applicable whenever memory or value based dependence information is available.
- A sound model to relax memory dependences with regards to reductions and its use in reduction-enabled polyhedral scheduling.
- A dependence based approach to identify vectorization and parallelization opportunities in the presence of reductions.

The remainder of this paper is organized as follows: We give a short introduction into the polyhedral model in Section 2. Thereafter, in Section 3, our reduction detection is described. Section 4 discusses the benefits and drawbacks of different reduction parallelization schemes, including privatization. Afterwards, we present different approaches to utilize the reduction properties in a polyhedral optimizer in Section 5. In the end we evaluate our work (Section 6), compare it to existing approaches (Section 7) and conclude with possible extensions in Section 8.

2. THE POLYHEDRAL MODEL

The main idea behind polyhedral loop nest optimizations is to abstract from technical details of the target program. Information relevant to the optimization goal is represented in a very powerful mathematical model and the actual optimizations are well understood transformations on this representation. In the context of optimization for data locality or parallelism, the relevant information is the iteration space of each statement, as well as the data dependences between individual statement instances.

```
    for (i = 0; i < NX; i++) {
R:    q[i] = 0;
      for (j = 0; j < NY; j++) {
S:      q[i] = q[i] + A[i][j] * p[j];
T:      s[j] = s[j] + r[i] * A[i][j];
      }
    }
```

Figure 2: BiCG Sub Kernel of BiCGStab Linear Solver.

Figure 2 shows an example program containing three statements R, S and T in a loop nest of depth two. Figure 3 shows the polyhedral representation of the individual iteration spaces for all statements, as well as value-based data dependences between individual instances thereof. R has a one-dimensional iteration space, as it is nested in the i-loop only. Statements S and T have a two-dimensional iteration space as they are nested in both the i-loop as well as in the j-loop. The axes in the Figure correspond to the respective loops. Single instances of each statement are depicted as dots in the graph. Dependences between individual statement instances are depicted as arrows: dashed ones for regular data dependences and dotted ones for loop carried data dependences.

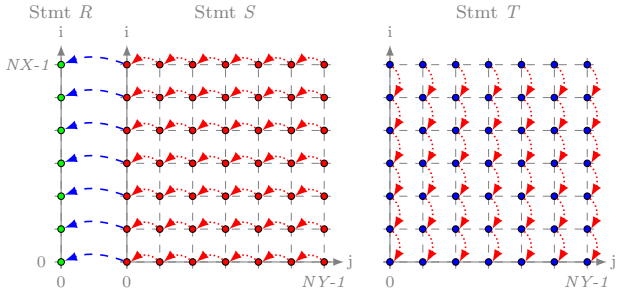

Figure 3: Polyhedral representation of statements R, S and T of the BiCG Sub Kernel of Figure 2.

In the polyhedral model the iteration space of a statement Q is represented as a multidimensional $\mathbb{Z}$-polytope $\mathcal{I}_Q$, defined by affine constraints on the iteration variables of loops surrounding the statement, as well as global parameters. The latter are basically loop invariant expressions like for example the upper bounds `NX` and `NY` of the loops in Figure 2. As a consequence, the polyhedral model is only applicable to well structured program parts with affine loop bounds and memory access functions, so called *Static Control Parts* (*SCoP*s) [8]. While, there are different over-approximations to increase the applicability (e.g., by Benabderrahmane [1]) we will assume that all restrictions of *SCoP*s are fulfilled.

The dependences between two statements Q and T are also represented as a multidimensional $\mathbb{Z}$-polytope known as the dependence polytope $\mathcal{D}_{<Q,T>}$. It contains a point $< i_Q, i_T >$ for every pair of instances $< i_Q > \in \mathcal{I}_Q$ and $< i_T > \in \mathcal{I}_T$ for which the latter depends on the former. To ease reading we will however omit the index of the dependence polytopes and only argue about the set of all dependences $\mathcal{D}$, defined as:

$$\mathcal{D} := \{< i_Q, i_T > \mid \forall Q, T \in SCoP : < i_Q, i_T > \in \mathcal{D}_{<Q,T>}\}$$

Later we will also distinguish all *Write-After-Write* (*WAW* or output dependence) dependences of $\mathcal{D}$ by writing $\mathcal{D}_{WAW}$.

A loop transformation in the polyhedral model is represented as an affine function θ_Q for each statement Q. It is often called scheduling or scattering function. This function translates a point in the original iteration space $\mathcal{I}_Q$ of statement Q into a new, transformed target space. One important legality criterion for such a transformation is that data dependences need to be respected: The execution of every instance of a source statement Q of a dependence has to precede the execution of the corresponding target statement T in the transformed space. Formulated differently: the target iteration vector of the value producing instance of Q has to be lexicographically smaller[1] than the target

[1]To compare two vectors of different dimensionality, we simply fill up the shorter vector with zeros in the end to match the dimensionality of the larger one.

iteration vector of the consuming instance of T:

$$< i_Q, i_T > \in \mathcal{D} \Rightarrow \theta_Q(i_Q) \ll \theta_T(i_T) \quad (1)$$

Multiple statements, or multiple instances of the same statement, that are mapped to the same point in the target space, can be executed in parallel. However, implementations of polyhedral schedulers [3, 29] usually generate scheduling functions with full rank, thus $rank(dom(\theta)) = rank(img(\theta))$. The parallelism is therefore not explicit in the scheduling function but is exposed later when the polyhedral representation is converted to target code.

There are two things that make the described model particularly interesting for loop transformation: First, unlike classical optimizers, a polyhedral optimizer does not only consider individual statements, but instead individual dynamic instances of each statement. This granularity leads to a far higher expressiveness. Second, the combination of multiple classical loop transformations, like for instance loop skewing, reversal, fusion, even tiling, typically used as atoms in a sequence of transformations, can be performed in one step by the scattering function. There is no need to come up with and evaluate different, possibly equivalent or even illegal combinations of transformations. Instead, linear optimization is used to optimize the scattering function for every individual statement with respect to an optimization goal.

3. DETECTING REDUCTIONS

Pattern based approaches on source statements are limited to find general reduction idioms [6, 21, 27]. The two main restrictions are the amount of patterns in the compiler's reduction pattern database and the sensitivity to the input code quality or preprocessing steps. To become as independent as possible of source code quality and canonicalization passes we replace the pattern recognition by a simple, data flow like analysis. This analysis will identify *reduction-like computations* within each polyhedral statement. Such a computation is a potential candidate for a reduction, thus it might be allowed to perform the computation in any order or even in parallel. Afterwards, we utilize the polyhedral dependence analysis [5] to precisely identify all reduction dependences [20] in a *SCoP*, hence to identify the actual reduction computations from the set of possible candidate (reduction-like) computations.

```
l = load A[f(i_S,p)]      store x A[f(i_S,p)]
                  x = y ⊙ z
```

Figure 5: SSA-based language subset.

The following discussion is restricted to the SSA-based language subset (`Insts`) depicted in Figure 5. Our implementation however handles all LLVM-IR [14] instructions.

The binary operation is parametrized with $\odot$ and can be instantiated with any arithmetic, bit-wise or logic binary operator. To distinguish associative and commutative binary operators we use $\oplus$ instead. The `load` instruction is applied to a memory location. It evaluates to the current value `x` stored in the corresponding memory location. The `store` instruction takes a value `x` and writes it to the given memory location. In both cases the memory location is described as `A[f(`i_S, p`)]`, where `A` is a constant array base pointer and `f(`i_S, p`)` is an affine function with regards to outer loop indices of the statement S (i_S) and parameters (p) of the *SCoP*. The range of a memory instruction is defined as the range of its affine access function:

$$ran(\texttt{store x A[f(}i_S,p\texttt{)]}) := ran(\texttt{A[f(}i_S,p\texttt{)]})$$
$$ran(\texttt{load A[f(}i_S,p\texttt{)]}) := ran(\texttt{A[f(}i_S,p\texttt{)]})$$
$$ran(\texttt{A[f(}i_S,p\texttt{)]}) := \texttt{A} + ran(\texttt{f(}i_S,p\texttt{)})$$

Note the absence of any kind of control flow producing or dependent instructions (ϕ instructions or branches). This is a side effect of the limited scope of the reduction detection analysis. It is applied only to polyhedral statements, in our setting basic blocks with exactly one `store` instruction. Furthermore, we assume all loop carried values are communicated in memory. This setup is equivalent to C source code statements without non-memory side effects.

3.1 Reduction-like Computations

Reduction-like computations are a generalization of the reduction definition used e.g., by Jouvelot [11] or Rauchwerger [21]. Their main characteristic is an associative and commutative computation which reduces a set of input values into reduction locations. Furthermore, the input values, the control flow and any value that might escape into a non-reduction location needs to be independent of the intermediate results of the reduction-like computation. The difference between reduction-like computations and reductions known in the literature is the restriction on other appearances of the reduction location in the loop nest. We do not restrict syntactic appearances of the reduction location base pointer as e.g., Rauchwerger [21] does, but only accesses to the actual reduction location in the same statement. This means a reduction-like computation on `A[i%2]` is not invalidated by any occurrence of `A[i%2 + 1]` in the same statement or any occurrence of `A` in another statement.

It is crucial to stress that we define reduction-like computations for a single polyhedral statement containing only a single `store`. Thus intermediate results of a reduction-like computation can only escape if they are used in a different statement or outside the *SCoP*. As we focus on memory reductions in a single statement we will assume such outside uses invalidate a candidate computation from being reduction-like. To this end we define the function:

$$hasOutsideUses : \texttt{Insts} \rightarrow bool$$

that returns true if an instruction is used outside its statement. In Section 3.3 we explain how the situation changes if multiple statements are combined into *compound statements* in order to save compile time.

Reconsider the array sum example in Figure 1. The reduction location is the variable `sum`, a scalar variable or zero dimensional array. However, we do not limit reduction-like computations to zero dimensional reduction locations, instead we allow multidimensional reduction locations, also called *histogram reductions* [18], as well. The second example, Figure 2, shows two such multidimensional reductions. The reduction locations are `q[i]` and `s[j]`. The first is variant in the outer loop, the second in the inner loop.

To detect reduction-like computations we apply the detection function t_S, shown in Figure 4, to the `store` in the polyhedral statement S. The idea is to track the flow of loaded values through computation up to the `store`. To this end, t_S(`I`) for any instruction `I` will assign each `load` a symbol that describes how the value loaded by `load` used up to and by `I`. We will use $\mathcal{R}_{op}$ to refer to the set of all

$$
\begin{array}{lcll}
t_S(\texttt{l = load A[f(}i_S,p\texttt{)]}) & := & \lambda l: & \text{if } (l \neq \texttt{l}) \text{ then } \bot \text{ else} \\
 & & & \text{if } (\mathit{hasOutsideUses}(l)) \text{ then } \top \text{ else } \uparrow \\
t_S(\texttt{x = y} \odot \texttt{z}) & := & \lambda l: & \text{if } (\{\,(t_S(y))(l),\ (t_S(z))(l)\,\} = \{\,\bot,\bot\,\}) \text{ then } \bot \text{ else} \\
 & & & \text{if } \neg(\mathit{isCommutative}(\odot) \wedge \mathit{isAssociative}(\odot)) \text{ then } \top \text{ else} \\
 & & & \text{if } (\mathit{hasOutsideUses}(x)) \text{ then } \top \text{ else} \\
 & & & \text{if } (\{\,(t_S(y))(l),\ (t_S(z))(l)\,\} = \{\,\uparrow,\bot\,\}) \text{ then } \odot \text{ else} \\
 & & & \text{if } (\{\,(t_S(y))(l),\ (t_S(z))(l)\,\} = \{\,\odot,\bot\,\}) \text{ then } \odot \text{ else } \top \\
t_S(\texttt{store x A[f(}i_S,p\texttt{)]}) & := & \lambda l: & \text{if } \texttt{x} \notin \texttt{Insts} \text{ then } \bot \text{ else} \\
 & & & \text{if } (\mathit{ran}(l) \cap \mathit{ran}(\texttt{A[f(}i_S,p\texttt{)]}) = \emptyset) \text{ then } \top \text{ else} \\
 & & & \text{if } (\exists l' : l \neq l' \wedge \mathit{ran}(l') \cap \mathit{ran}(l) \cap \mathit{ran}(\texttt{A[f(}i_S,p\texttt{)]}) \neq \emptyset) \text{ then } \top \text{ else } (t_S(\texttt{x}))(l)
\end{array}
$$

Figure 4: Detection function for reduction-like computations: $t_S : \texttt{Insts} \to (\mathit{loads}(S) \to \mathcal{R}_{op})$.

four symbols. It includes the $\bot$ indicating that the `load` was not used by the instruction, the $\uparrow$ to express that it was only loaded but not yet used in any computation, the $\top$ stating that the loaded value may have been used in a non-associative or non-commutative computation. Additionally, the $\oplus$ is used when the loaded value was exactly one input of a chain of $\oplus$ operations. Note that only a `load l` that flows with $\oplus$ into the `store` is a valid candidate for a reduction-like computation and only if the load and the `store` access (partially) the same memory. Furthermore, we forbid all other `load` instructions in the statement to access the same memory as both `l` and the `store` as that would again make the computation potentially non-associative and non-commutative.

If a valid `load l` was found, it is the unique `load` instruction inside the statement S that accesses (partially) the same memory as the `store s` and $(t_S(\texttt{s}))(\texttt{l})$ is an associative and commutative operation $\oplus$. We will refer to the quadruple (S, `l`, $\oplus$, `s`) as the reduction-like computation R_c of S and denote the set of all reduction-like computations in a *SCoP* as $\mathcal{R}_c$.

It is worth noting that we explicitly allow the access functions of the `load` and the `store` to be different as for example shown in Figure 6. In such cases a reduction can manifest only for certain parameter valuations or, as shown, for certain valuations of outer loop indices. Additionally, we could easily extend the definition to allow non-affine but Presburger accesses or even over-approximated non-affine accesses if they are pure. It is also worth to note that our definition does not restrict the shape of the induced reduction dependences.

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    A[j] = A[j-i] + Mat[i][j];
```

Figure 6: Conditional reduction with different access functions.

3.2 Reduction Dependences

While the data flow analysis performed on all polyhedral statements only marks reduction-like computations, we are actually interested in *reduction dependences* [20]. These loop carried self dependences start and end in two instances of the same reduction-like computation and they inherit some properties of this computation. Similar to the reduction-like computation, reduction dependences can be considered to be "associative" and "commutative". The latter allows a schedule to reorder the iterations participating in the reduction-like computation while it can still be considered valid, however all non-reduction dependences still need to be fulfilled.

We split the set of all dependences $\mathcal{D}$ into the set of reduction dependences $\mathcal{D}_\rho \subseteq \mathcal{D}$ and the set of non-reduction dependences $\mathcal{D}_\nu := \mathcal{D} \setminus \mathcal{D}_\rho$. Now we can express the commutativity of a reduction dependence by extending the causality condition given in Constraint 1 as follows:

$$< i_Q, i_T > \in \mathcal{D}_\nu \implies \theta_Q(i_Q) \ll \theta_T(i_T) \tag{2}$$

$$< i_Q, i_T > \in \mathcal{D}_\rho \implies \theta_Q(i_Q) \neq \theta_T(i_T) \tag{3}$$

Constraint 2 is the same as the original causality condition (Constraint 1), except that we restrict the domain to non-reduction dependences $\mathcal{D}_\nu$. For the remaining reduction dependences $\mathcal{D}_\rho$, Constraint 3 states that the schedule θ can reorder two iterations freely, as long as they are not mapped to the same time stamp. However, relaxing the causality condition for reduction dependences is only valid if $\mathcal{D}$ contains all transitive reduction dependences. This is for example the case if $\mathcal{D}$ is computed by a memory-based dependence analysis. In case only value-based dependence analysis [5] was performed it is also sufficient to provide the missing transitive reduction dependences e.g., by recomputing them using a memory-based dependence analysis.

Reconsider the BiCG kernel (Figure 2) and its non transitive (value-based) set of dependences $\mathcal{D}$ shown in Figure 3. If we remove all reduction dependences $\mathcal{D}_\rho$ from $\mathcal{D}$, the only constraints left involve statement R and the iterations of statements S with $j = 0$. Consequently, there is no reason not to schedule the other instances of statement S before statement R.

To address the issue of only value-based dependences without recomputing memory-based ones we use the transitive closure $\mathcal{D}_\rho^+{}_S$ of the reduction dependences for a statement S (Equation 4). As the transitive closure of a Presburger relation is not always a Presburger relation we might have to use an over-approximation to remain sound, however Pugh and Wonnacot [19] describe how the transitive closure can also be computed precisely for exact direction/distance vectors. They also argue in later work [20] that the transitive closure of value-based reduction dependences of real programs can be computed in an easy and fast way.

If we now interpret $\mathcal{D}_\rho^+{}_S$ as a relation that maps instances of a reduction statement S to all instances of S transi-

tively dependent, we can define *privatization dependences* $\mathcal{D}_\tau$ (Equation 5). In simple terms, $\mathcal{D}_\tau$ will ensure that no non-reduction statement accessing the reduction location can be scheduled in-between the reduction statement instances by extending the dependences ending or starting from a reduction access to all reduction access instances. This also means that in case no memory locations are reused e.g., after renaming and array expansion [4] was applied, the set of privatization dependences will be empty.

$$\mathcal{D}_\rho^+{}_S := (\mathcal{D}_\rho \cap < \mathcal{I}_S, \mathcal{I}_S >)^+ \quad (4)$$

$$\mathcal{D}_\tau := \{< i_T, \mathcal{D}_\rho^+{}_S(i_S) > \mid < i_T, i_S > \in \mathcal{D}_{<T,S>}\} \cup \{< \mathcal{D}_\rho^+{}_S(i_S), i_T > \mid < i_S, i_T > \in \mathcal{D}_{<S,T>}\} \quad (5)$$

Privatization dependences overestimate the dependences that manual privatization of the reduction locations would cause. They are used to create alternative causality constraint for the reduction statements that enforce the initial order between the reduction-like computation and any other statement accessing the reduction locations. To make use of them we replace Constraint 2 by Constraint 6.

$$< i_Q, i_T > \in (\mathcal{D}_\nu \cup \mathcal{D}_\tau) \implies \theta_Q(i_Q) \ll \theta_T(i_T) \quad (6)$$

If we now utilize the associativity of the reduction dependences we can compute intermediate results in any order before we combine them to the final result. As a consequence we can allow parallelization of the reduction-like computation, thus omit Constraint 3; *thereby eliminating the reduction dependences $\mathcal{D}_\rho$ from the causality condition of a schedule completely.* However, parallel execution of iterations connected by reduction dependences requires special "treatment" of the accesses during the code generation as described in Section 4.

The restriction on polyhedral statements, especially that it contains at most one `store` instruction, eases the identification of reduction dependences; they are equal to all loop carried *Write-After-Write* self dependences over a statement with a reduction-like computation[2]. Thus, $\mathcal{D}_\rho$ can be expressed as stated in Equation 7.

$$\mathcal{D}_\rho := \mathcal{D}_{WAW} \cap \{ \mathcal{I}_S \times \mathcal{I}_S \mid (S, \mathtt{l}, \oplus, \mathtt{s}) \in \mathcal{R}_c\} \quad (7)$$

3.3 General Polyhedral Statements

Practical polyhedral optimizer operate on different granularities of polyhedral statements; a crucial factor for both compile time and quality of the optimization. While *Clan* [3] operates on C statements, *Polly* is based on basic blocks in the SSA-based intermediate language of LLVM. The former eases not only reduction handling but also offers more scheduling freedom. However, the latter can accumulate the effects of multiple C statements in one basic block, thus it can perform better with regards to compile time. Finding a good granularity for a given program, e.g., when and where to split a LLVM basic block in the Polly setting, is a research topic on its own but we do not want to limit our approach to one fixed granularity. Therefore, we will now assume a polyhedral statement can contain multiple `store` instructions, thus we allow arbitrary basic blocks.

As a first consequence we have to check that intermediate values of a reduction-like computation do not escape into non-reduction memory locations. This happens if and only if intermediate values—and therefore the reduction `load`—flow into multiple `store` instructions of the statement S. Additionally, other `store` instructions are not allowed to override intermediate values of the reduction computation. Thus, $(S, \mathtt{l}, \oplus, \mathtt{s})$ can only be a reduction-like computations, if for all other `store` instructions $\mathtt{s'}$ in S:

$$(t(\mathtt{s'}))(\mathtt{l}) = \bot \ \wedge \ range(\mathtt{s'}) \cap range(\mathtt{s}) \cap range(\mathtt{l}) = \emptyset$$

Furthermore, we cannot assume that all loop carried *WAW* self dependences of a statement containing a reduction-like computation are reduction dependences: other read and write accesses contained in the statement could cause the same kind of dependences. However, we are particularly interested in dependences caused by the `load` and `store` instruction of a reduction-like computation R_c. To track these accesses separately we can pretend they are contained in their own statement S_{R_c} that is executed at the same time as S (in the original iteration space). This is only sound as long as no other instruction in S accesses (partially) the same memory as the `load` or the `store`, but this was already a restriction on reduction-like computations. The definition of reduction dependences (Equation 7) is finally changed to:

$$\mathcal{D}_\rho := \mathcal{D}_{WAW} \cap \{ \mathcal{I}_{S_{R_c}} \times \mathcal{I}_{S_{R_c}} \mid R_c \in \mathcal{R}_c\} \quad (8)$$

It is important to note the increased complexity of the dependence detection problem when we model reduction accesses separately. However, our experiments in Section 6 show that the effect is (in most cases) negligible. Furthermore, we want to stress that this kind of separation is not equivalent to separating the reduction access at the statement level as we do not allow separate scheduling functions for S and S_{R_c}. Similar to a fine-grained granularity at the statement level, separation might be desirable in some cases, however it suffers from the same drawbacks.

4. PARALLEL EXECUTION

When executing accesses to a reduction location x, p times in parallel, it needs to be made sure that the read-modify-write cycle on x happens atomically. While doing exactly that — performing atomic read-modify-write operations — might be a viable solution in some contexts [31], it is generally too expensive. The overhead of an atomic operation easily outweighs the actual work for smaller tasks [18]. Additionally, the benefit of vectorization is lost for the reduction, as atomic operations have to scalarize the computation again. We will therefore focus our discussion and the evaluation on privatization as it is generally well-suited for the task at hand [18].

4.1 Privatizing Reductions

Privatization means that every parallel context c_i, which might be a thread or just a vector lane, depending on the kind of parallelization, gets its own private location x_i for x. In front of the parallelized loop carrying a reduction dependence p, private locations $x_1, \cdots, x_p$ of x are allocated and initialized with the identity element of the corresponding reduction operation $\oplus$. Every parallel context c_i now non-atomically, and thus cheaply, modifies its very own location x_i. After the loop, but before the first use of the x, accumulation code needs to join all locations into x again, thus: $x := x \oplus x_1 \oplus \cdots \oplus x_p$.

[2]In this restricted environment we could also use the *Read-After-Write* (RAW) dependences instead of the WAW ones.
[3]http://icps.u-strasbg.fr/~bastoul/development/clan/

```
// (A) init
for (i = 0; i < NX; i++)
  // (B) init
  for (j = 0; j < NY; j++)
    // (C) init
    for (k = 0; k < NZ; k++)
      P[j] += Q[i][j] * R[j][k];
    // (C) aggregate
  // (B) aggregate
// (A) aggregate
```

Figure 7: Possible privatization locations (A-C) for the reduction over `P[j]`.

Such a privatization transformation is legal due to the properties of a reduction operation. Every possible user of x sees the same result after the final accumulation has been performed as it would have seen before the transformation. Nevertheless we gained parallelism which cannot be exploited without the reduction properties. It might seem, that the final accumulation of the locations needs to be performed sequentially, but note that the number of locations does not necessarily grow with the problem size but instead only with the maximal number of parallel contexts. Furthermore, accumulation can be done in logarithmic time by parallelizing the accumulation correspondingly [6].

One positive aspect of using privatization to fix a broken reduction dependence is that it is particularly well-suited for both ways of parallelization usually performed in the polyhedral context: thread parallelism and vectorization. For thread parallelism real private locations of the reduction address are allocated; in case of vectorization, a vector of suitable width is used.

As described, privatization creates "copies" of the reduction location, one for each instance possibly executed in parallel. While we can limit the number of private locations (this corresponds to the maximal number of parallel contexts), we cannot generally bound the number of reduction locations. Furthermore, the number of necessary locations, as well as the number of times initialization and aggregation is needed, varies with the placement of the privatization code.

Consider the example in Figure 7. Different possibilities exist to exploit reduction parallelism: using placement C for the privatization, the k-loop could be executed in parallel and only p private copies of the reduction location are necessary. There is no benefit in choosing location B as we then need $p \times NY$ privatization locations (we have NY different reduction locations modified by the j-loop and p parallel contexts), but there is no gain in the amount of parallelism (the j-loop is already parallel). Finally, choosing location A for privatization might be worthwhile. We still only need $p \times NY$ privatized values, but save aggregation overhead: While for location C, p values are aggregated $NX \times NY$ times and for location B, $p \times NY$ locations are aggregated NX times, for location A, $p \times NY$ locations are aggregated only once. Furthermore, the i-loop can now be parallelized.

In general, a trade-off has to be made between memory consumption, aggregation time and exploitable parallelism. Finding a good placement however is difficult and needs to take the optimization goal, the hardware and the workload size into account. Furthermore, depending on the scheduling, the choices for privatization code placement in the resulting code might be limited, which suggests that the scheduler should be aware of the implications of a chosen schedule with respect to the efficiency of necessary privatization.

In Section 6.1 we discuss the effect of different placement choices for the BiCG benchmark shown in Figure 2.

5. MODELING REDUCTIONS

As mentioned earlier, the set $\mathcal{D}$ of all dependences is partitioned into the set $\mathcal{D}_\rho$ of reduction induced dependences and $\mathcal{D}_\nu$ of regular dependences. Reduction dependences inherit properties similar to associativity and commutativity from the reduction operator $\oplus$: the corresponding source and target statement instances can be executed in any order—provided $\oplus$ is a commutative operation—or in parallel—if $\oplus$ is at least associative. In order to exploit these properties the polyhedral optimizer needs to be aware of them. To this end we propose different scheduling and code generation schemes.

Reduction-Enabled Code Generation
: is a simple, non-invasive method to realize reductions during the code generation, thus without modification of the polyhedral representation of the *SCoP*.

Reduction-Enabled Scheduling
: exploits the properties of reductions in the polyhedral representation. All reduction dependences are basically ignored during scheduling, thereby increasing the freedom of the scheduler.

Reduction-Aware Scheduling
: is the representation of reductions and their realization via privatization in the polyhedral optimization. The scheduler decides when and where to make use of reduction parallelism. However, non-trivial modifications of the polyhedral representation and the current state-of-the-art schedulers are necessary.

5.1 Reduction-Enabled Code Generation

The reduction-enabled code generation is a simple, non-invasive approach to exploit reduction parallelism. The only changes needed to enable this technique are in the code generation, thus the polyhedral representation is not modified. So far, dimensions or loops are marked parallel if they do not carry any dependences. With regards to reduction dependences we can relax this condition, hence we can mark non-parallel dimensions or loops as parallel, provided we add privatization code, if they only carry reduction dependences. To implement this technique we add one additional check to the code generation that is executed for each non-parallel loop of the resulting code that we want to parallelize. It uses only non-reduction dependences $\mathcal{D}_\nu$ not $\mathcal{D}$ to determine if the loop exclusively carries reduction dependences. If so, the reduction locations corresponding to the broken dependences are privatized and the loop is parallelized.

Due to its simplicity, it is easily integrable into existing optimizers while the compile time overhead is reasonably low. However, additional heuristics are needed. First, to decide if reductions should be realized e.g., if privatization of a whole array is worth the gain in parallelism. And second, where the privatization statements should be placed (cf. Section 4.1). Note that usually the code generator has no, and in fact should not have any, knowledge of the optimization goal of the scheduler.

Apart from the need for heuristics, reduction-aware code generation also misses opportunities to realize reductions effectively. This might happen if the scheduler has no reason to perform an enabling transformation or the applied transformation even disabled the reduction. Either way, it is hard to predict the outcome of this approach.

5.2 Reduction-Enabled Scheduling

In contrast to reduction-aware code generation, which is basically a post-processing step, reduction-enabled scheduling actually influences the scheduling processes by eliminating reduction dependences beforehand. Therefore, the scheduler is (1) unaware of the existence of reductions and their dependences and (2) has more freedom to schedule statements if they contain reduction instances. While this technique allows to exploit reductions more aggressively, there are still disadvantages. First of all, this approach relies on reduction-aware code generation as a back-end, hence it shares the same problems. Second, the scheduler's unawareness of reduction dependences prevents it from associating costs to reduction realization. Thus, privatization is implicitly assumed to come for free. Consequently, the scheduler does not prefer existing, reduction-independent parallelism over reduction parallelism and therefore may require unnecessary privatization code.

For the BiCG example (Figure 2) omitting the reduction dependences might not result in the desired schedule if we assume we are only interested in one level of outermost parallelism[4] and furthermore that the statements S and T have been split prior to the scheduling. In this situation we want to interchange the outer two loops for the T statement in order to utilize the inherent parallelism, not the reduction parallelism. However, without the reduction dependences the scheduler will not perform this transformation. In order to decrease the severity of this problem, the reduction dependences can still be used in the proximity constraints of the scheduler [29], thus the scheduler will try to minimize the dependence distance between reduction iterations and implicitly move them to inner dimensions. This solves the problem for all Polybench benchmarks with regards to outermost parallelism, however it might negatively affect vectorization if e.g., the innermost parallel dimension is always vectorized.

5.3 Reduction-Aware Scheduling

Reduction-enabled scheduling results in generally good schedules for our benchmark set, however resource constraints as well as environment effects, both crucial to the overall performance, are not represented in the typical objective function used by polyhedral optimizers. In essence we believe, the scheduler should be aware of reductions and the cost of their privatization, in terms of memory overhead as well as aggregation costs. This is especially true if the scheduler is used to decide which dimensions should be executed in parallel or if there are tight memory bounds (e.g., on mobile devices).

In Section 6.1 we show that the execution environment as well as the values of runtime parameters are crucial factors in the actual performance of parallelized code, even more when reductions are involved. While a reduction-aware scheduler could propose different parallelization schemes for different execution environments or parameter values, there is more work needed in order to (1) predict the effects of parallelization and privatization on the actual platform and to (2) express them as affine constraints in the scheduling objective function.

[4]A reasonable assumption for desktop computers or moderate servers with a low number of parallel compute resources.

Parallel	$2^{10} \times 2^{10}$		$2^{12} \times 2^{12}$		$2^{14} \times 2^{14}$		$2^{15} \times 2^{15}$	
Outer	0.19	0.55	2.31	0.75	3.91	0.72	2.19	0.96
Tile	0.03	1.10	0.32	1.54	0.10	1.60	0.16	2.21

Table 1: BiCG run-time results. The values are speedups compared to the sequential *Polly* version, first for the 32-core machine, then for the 4-core machine.

6. EVALUATION

We implemented Reduction-Enabled Scheduling (c.f., Section 5.2) in the polyhedral optimizer *Polly* and evaluated the effects on compile time and run-time on the Polybench 3.2. We used an *Intel(R) core i7-4800MQ* quad core machine and the standard input size of the benchmarks.

Our approach is capable of identifying and modeling all reductions as described in Section 3: in total 52 arithmetic reductions in 30 benchmarks [5].

As described earlier, our detection virtually splits polyhedral statements to track the effects of the `load` and `store` instructions that participate in reduction-like computations. As this increases the complexity of the performed dependence analysis we timed this particular part of the compilation for each of the benchmarks and compared our **??** hybrid dependence analysis to a completely **??** access-wise analysis and the default **??** statement-wise one. We use the term hybrid because reduction accesses are tracked separately while other accesses are accumulated on statement level.

As shown in Figure 8 (top) our approach takes up to 5× as long (benchmark lu) than the default implementation but in average only 85% more. Access-wise dependence computation however is up to 10× slower than the default and takes in average twice as long as our hybrid approach. Note that both approaches do not only compute the dependences (partially) on the access level but also the reduction and privatization dependences as explained in Section 3.2.

Figure 8 (bottom) shows the speedup of our approach compared to the non-reduction *Polly*. The additional scheduling freedom causes speedups for the data-mining applications (correlation and covariance) but slowdowns especially for the matrix multiplication kernels (2mm, 3mm and gemm). This is due to the way *Polly* generates vector code. The deepest dimension of the new schedule that is parallel (or now reduction parallel) will be strip-mined and vectorized. Hence the stride of the contained accesses, crucial to generate efficient vector code, is not considered. However, we do not believe this to be a general shortcoming of our approach as there are existing approaches to tackle the problem of finding a good vector dimension [13] that would benefit from the additional scheduling freedom as well as the knowledge of reduction dependences.

6.1 BiCG Case Study

Polybench is a collection of inherent parallel programs, there is only one—the BiCG kernel— that depends on re-

[5]This assumes the benchmarks are compiled with *-ffast-math*, otherwise reductions over floating point computations are not detected.

duction parallelism. To study the effects of parallelization combined with privatization of multidimensional reductions in the BiCG kernel we compared two parallel versions to the non-parallel code *Polly* would generate without reduction support. The first version "*Outer*" has a parallel outermost loop and therefore needs to privatize the whole array `s`. The second version "*Tile*" parallelizes the second outermost loop. Due to tiling, only "tile size" (here 32) locations of the `q` array need to be privatized. Table 1 shows the speedup compared to the sequential version for both a quad core machine and a 8×4-core server. As the input grows larger the threading overhead as well as the inter-chip communication on the server will cause the speedup of *Tile* to stagnate, however on a one chip architecture this version generally performs best. *Outer* on the other hand will perform well on the server but not on the 4-core machine. We therefore believe the environment is a key factor in the performance of reduction-aware parallelization and a reduction-aware scheduler is needed to decide under which run-time conditions privatization becomes beneficial.

7. RELATED WORK

Reduction aware loop parallelization has been a long lasting research topic. Different approaches to detect reduction, to model them and finally to optimize them have been proposed. As our work has some intersection with all three parts we will discuss them in separation.

7.1 Detection

Reduction detection started with pattern based approaches on source statements [11, 17, 22, 18, 21, 24] and evolved to more elaborate techniques that use symbolic evaluation [6], a data dependency graph [27] or even a program dependency graph [17] to find candidates for reduction computations.

For functional programs Xu et al. [30] use a type system to deduce parallel loops including pattern based reductions. Their typing rules are similar to our detection function (Figure 4) we use to identify reduction-like computations.

Sato and Iwasaki [25] describe a pragmatic system to detect and parallelize reduction and scan operations based on the ideas introduced by Matsuzaki et al. [15]: the representation of (part of) the loop as a matrix multiplication with a state vector. They can handle mutually recursive scan and reduction operations as well as maximum computations implemented with conditionals, but they are restricted to innermost loops and scalar accumulation variables. As an extension Zou and Rajopadhye [32] combined the work with the polyhedral model and the recurrence detection approach of Redon and Feautrier [22, 24]. This combination overcomes many limitations, e.g., multidimensional reductions (and scans) over arrays are handled. However, the applicability is still restricted to scans and reductions representable in State Vector Update Form [12].

In our setting we identify actual reductions utilizing the already present dependence analysis, an approach very similar to the what Suganuma et al. [27] proposed to do. However, we only perform the expensive, access-wise dependence analysis for reduction candidates, and not for all accesses in the *SCoP*. Nevertheless, both detections do not need the reductions to be isolated in a separate loop as assumed by Fisher and Ghuloum [6] or Pottenger and Eigenmann [18]. Furthermore, we allow the induced reduction dependences to be of any form and carried by any subset of outer loop dimensions. This is similar to the nested *Recur* operator introduced by Redon and Feautrier [22, 24]. Hence, reductions are not only restricted to a single loop dimension, as in other approaches [11, 6, 25], but can also be multidimensional as shown in Figure 2.

7.2 Modeling

Modeling reductions was commonly done implicitly, e.g., by ignoring the reduction dependences during a post parallelization step [11, 17, 22, 18, 21, 30, 28]. This is comparable to the reduction-enabled code generation described in Section 5.1. However, we believe the full potential of reductions can only be exposed when the effects are properly modeled on the dependence level.

The first to do so, namely to introduce reduction dependences, where Pugh and Wonnacot [20]. Similar to most other approaches [22, 23, 27, 24, 7, 25, 32], the detection and modeling of the reduction was performed only on C-like statements and utilizing a precise but costly access-wise dependence analysis (see the upper part of Figure 8). In their work they utilize both memory and value-based dependence information to identify statements with an iteration space that can be executed in parallel, possibly after transformations like array expansion. They start with the memory-based dependences and compute the value-based dependences as well as the transitive self-dependence relation for a statement in case the statement might not be inherently sequential.

Stock et al.[26] describe how reduction properties can be exploited in the polyhedral model, however neither do they describe the detection nor how omitting reduction dependences may affect other statements.

In the works of Redon and Feautrier [23] as well as the extension to that by Gupta et al. [9] the reduction modeling is performed on SAREs on which array expansion [4] and renaming was applied, thus all dependences caused by memory reuse were eliminated. In this setting the possible interference between reduction computation and other statements is simplified but it might not be practical for general purpose compilers due to memory constraints. As an extension to these scheduling approaches on SAREs we introduced privatization dependences. They model the dependences between a reduction and the surrounding statements without the need for any special preprocessing of the input. However, we still allow polyhedral optimizations that will not only affect the reduction statement but all statements in a *SCoP*.

7.3 Optimization

Optimization in the context of reductions is twofold. There is the parallelization of the reduction as it is given in the input and the transformation as well as possible parallelization of the input with awareness of the reduction properties. The first idea is very similar to the reduction-enabled code generation as described in Section 5.1. In different variations, innermost loops [25], loops containing only a reduction [6, 18] or recursive functions computing a reduction [30] were parallelized or replaced by a call to a possibly parallel reduction implementation [28]. The major drawback of such optimizations is that reductions have to be computed either in isolation or with the statements that are part of the source loop that is parallelized. Thus, the reduction statement instances are never reordered or interleaved with

other statement instances, even if it would be beneficial. In order to allow powerful transformations in the context of reductions, their effect, hence the reduction dependences, as well as their possible interactions with all other statement instances must be known. The first polyhedral scheduling approach by Redon and Feautrier [23] that optimally[6] schedules reduction together with other statements assumed reductions to be computable in one time step. With such atomic reduction computations there are no reduction statement instances that could be reordered or interleaved with other statement instances. Gupta et al. [9] extended that work and lifted the restriction on an atomic reduction computation. As they schedule the instances of the reduction computation together with the instances of all other statements their work can be seen as a reduction-enabled scheduler that optimally minimizes the latency of the input.

To speed up parallel execution of reductions the runtime overhead needs to be minimized. Pottenger [18] proposed to privatize the reduction locations instead of locking them for each access and Suganuma et al. [27] described how multiple reductions on the same memory location can be coalesced. If dynamic reduction detection [21] was performed, different privatization schemes to minimize the memory and runtime overhead were proposed by Yu et al. [31]. While the latter is out of scope for a static polyhedral optimizer, the former might be worth investigating once our approach is extended to multiple reductions on the same location.

In contrast to polyhedral optimization or parallelization, Gautam and Rajopadhye [7] exploited reduction properties in the polyhedral model to decrease the complexity of a computation in the spirit of dynamic programming. Their work on reusing shared intermediate results of reduction computations is completely orthogonal to ours.

While Array Expansion, as introduced by Feautrier [4], is not a reduction optimization, it is still similar to the privatization step of any reduction handling approach. However, the number of privatization copies the approach introduces, the accumulation of these private copies as well as the kind of dependences that are removed differ. While privatization only introduces a new location for each processor or vector lane, general array expansion introduces a new location for each instance of the statement. In terms of dependences, array expansion aims to remove false output and anti dependences that are introduced by the reuse of memory while reduction handling approaches break output and flow dependences that are caused by a reduction computation. Because of the flow dependences—the actual reuse of formerly computed values—the reduction handling approaches also need to implement a more elaborate accumulation scheme that combines all private copies again.

8. CONCLUSIONS AND FUTURE WORK

Earlier work already utilized reduction dependences in different varieties, depending on how powerful the detection was. Whenever reductions have been parallelized the reduction dependences have been implicitly ignored, in at least two cases they have even been made explicit [20, 26]. However, to our knowledge, we are the first to add the concept of privatization dependences in this context. The reason is simple: we believe the parallel execution of a loop containing a reduction is not always the best possible optimization. Instead we want to allow any transformation possible to our scheduler with only one restriction: the integrity of the reduction computation needs to stay intact. In other words, no access to the reduction location is scheduled between the first and last instance of the reduction statement. This allows our scheduler not only to optimize the reduction statement in isolation, but also to consider other statements at the same time without the need for any preprocessing to get a SARE-like input.

To this end we presented a powerful reduction detection based on computation properties and the polyhedral dependence analysis. Our design leverages the power of polyhedral loop transformations and exposes various optimization possibilities including parallelism in the presence of reduction dependences. We showed how to model and leverage associativity and commutativity to relax the causality constraints and proposed three approaches to make polyhedral loop optimization reduction-aware. We believe our framework is the first step to handle various well-known idioms, e.g., privatization or recurrences, not yet exploited in most practical polyhedral optimizers.

Furthermore, we showed that problems and opportunities arising from reduction parallelism (see Section 6.1) have to be incorporated into the scheduling process, thus the scheduling in the polyhedral model needs to be done in a more realistic way. The overhead of privatization and the actual gain of parallelism are severely influenced by the execution environment (e.g., available resources, number of processors and cores, cache hierarchy), however these hardware specific parameters are often not considered in a realistic way during the scheduling process.

Extensions to this work include a working reduction-aware scheduler and the modeling of multiple reduction-like computations as well as other parallelization preventing idioms. In addition we believe that a survey about the applicability of different reduction detection schemes as well as optimization approaches in a realistic environment is needed. In any case this would help us to understand reductions not only from the theoretical point of view but also from a practical one.

[6] e.g., according to the latency

9. ACKNOWLEDGMENTS

We would like to thank Tobias Grosser, Sebastian Pop and Sven Verdoolaege for the helpful discussions during the development and implementation of this approach. Furthermore, we want to thank the reviewers who not only provided extensive comments on how to further improve this work but also pointed us to related work that was previously unknown to us. Lastly, we would like to thank Tomofumi Yuki for giving us many helpful tips.

10. REFERENCES

[1] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11):1526–1538, Nov. 1989.

[3] U. Bondhugula, M. M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan.

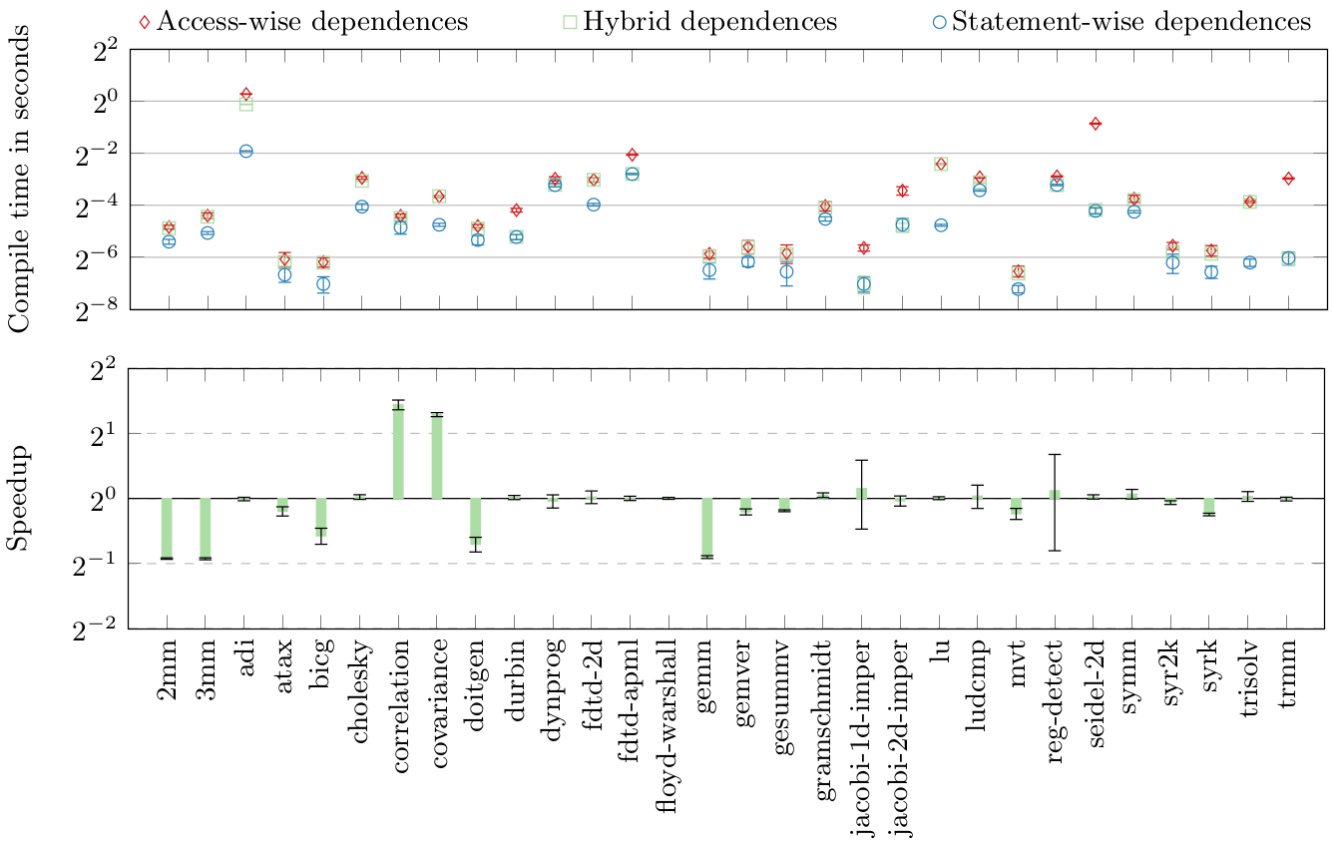

Figure 8: Evaluation results for Polybench 3.2. In the upper part the compile time for different grained dependence analyses is shown, in the lower part the speedup of Polly with reduction support compared to Polly without reduction support.

Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. ETAPS '08.

[4] P. Feautrier. Array expansion. In *Proceedings of the 2Nd International Conference on Supercomputing*, ICS '88, pages 429–441, New York, NY, USA, 1988. ACM.

[5] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[6] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 135–146, New York, NY, USA, 1994. ACM.

[7] Gautam and S. Rajopadhye. Simplifying reductions. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 30–41, New York, NY, USA, 2006. ACM.

[8] T. Grosser, A. Größlinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[9] G. Gupta, S. Rajopadhye, and P. Quinton. Scheduling reductions on realistic machines. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 117–126, New York, NY, USA, 2002. ACM.

[10] P. Jouvelot. Parallelization by semantic detection of reductions. In *Proc. Of the European Symposium on Programming on ESOP 86*, pages 223–236, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[11] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd International Conference on Supercomputing*, ICS '89, pages 186–194, New York, NY, USA, 1989. ACM.

[12] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.*, 22(8):786–793, Aug. 1973.

[13] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, New York, NY, USA, 2013. ACM.

[14] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[15] K. Matsuzaki, Z. Hu, and M. Takeichi. Towards automatic parallelization of tree reductions in

dynamic programming. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 39–48, New York, NY, USA, 2006. ACM.

[16] S. P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Synthesis Lectures on Computer Architecture. 2012.

[17] S. S. Pinter and R. Y. Pinter. Program optimization and parallelization using idioms. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 79–92, New York, NY, USA, 1991. ACM.

[18] B. Pottenger and R. Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 444–448, New York, NY, USA, 1995. ACM.

[19] W. Pugh. Uniform techniques for loop optimization. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, pages 341–352, New York, NY, USA, 1991. ACM.

[20] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Program. Lang. Syst.*, 16(4):1248–1278, July 1994.

[21] L. Rauchwerger and D. Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM.

[22] X. Redon and P. Feautrier. Detection of recurrences in sequential programs with loops. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '93, pages 132–145, London, UK, UK, 1993. Springer-Verlag.

[23] X. Redon and P. Feautrier. Scheduling reductions. In *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, pages 117–125, New York, NY, USA, 1994. ACM.

[24] X. Redon and P. Feautrier. Detection of scans in the polytope model. *Parallel Algorithms Appl.*, 15(3-4):229–263, 2000.

[25] S. Sato and H. Iwasaki. Automatic parallelization via matrix multiplication. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 470–479, New York, NY, USA, 2011. ACM.

[26] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 65–76, New York, NY, USA, 2014. ACM.

[27] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th International Conference on Supercomputing*, ICS '96, pages 18–25, New York, NY, USA, 1996. ACM.

[28] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA, 2014. ACM.

[29] S. Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[30] D. N. Xu, S.-C. Khoo, and Z. Hu. Ptype system: A featherweight parallelizability detector. In *In Proceedgins of 2nd Asian Symposium on Programming Languages and Systems (APLAS 2004), LNCS 3302*, pages 197–212. Springer, LNCS, 2004.

[31] H. Yu, D. Zhang, and L. Rauchwerger. An adaptive algorithm selection framework. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 278–289, Washington, DC, USA, 2004. IEEE Computer Society.

[32] Y. Zou and S. Rajopadhye. Scan detection and parallelization in "inherently sequential" nested loop programs. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 74–83, New York, NY, USA, 2012. ACM.